
User's Guide for MulTEx - The Multi Tier Exception Handling Framework

Christoph Knabe, - Beuth University of Applied Sciences Berlin -

2011-11-08

Table of Contents

Introduce Central Exception Reporting	1
Reporting in Command Line Applications	2
Reporting in Swing Applications	2
Reporting in JSP Applications	3
Reporting in background threads	6
How to introduce/use MulTEx in the API layers	6
What is a MulTEx exception?	7
How to declare a MulTEx exception with an internationalizable message text pattern	7
How to implement negative business rules by throwing Exc-objects	9
How to implement exception chaining by throwing a Failure object	10
How to cope with many Failure causes (list exceptions, exception treeing)	11

The main goal of this user's guide is to tell, how to quickly introduce robust and diagnostic exception handling and reporting into a Java software system.

These hints are easier to follow, if you are writing a software system from scratch, but can be applied onto existing software, as well.

You should introduce it according to these priorities:

1. introduce central exception reporting,
2. assure exception propagation to the highest possible level,
3. provide all exceptions with diagnostic parameters,
4. provide natural message texts for all exceptions.

Introduce Central Exception Reporting

When you are using MulTEx, the main paradigm for error handling and reporting are Java exceptions. Any misfunction will be passed to the caller of a method by the standard Java means, an exception. This will be propagated to the upmost level, where it still can be caught, in order to report it. Handling of an individual exception should rarely be necessary.

Depending on the type of user interface, there are different places, where you can catch all propagated exceptions. Sometimes it is not simple to find this place in the UI documentation. Sometimes you will not be able to find it, and must use a workaround in order to assure central exception reporting.

Reporting in Command Line Applications

In a Java application, invoked as a command line, you should report, both the exception's messages, and its stack trace, as the command line application typically does not have a button for requesting the details of the stack trace. The destination for the exception reporting usually will be `System.err`.

If you specify the main method as

```
public static void main(final String[] i_args) throws Exception {...}
```

you will already have a centralized exception reporting, but the Java Runtime system will report only the stack trace without giving nice textual error messages.

In order to get the textual message chain, and the stack trace, you must write your main method as follows:

```
public static void main(final String[] i_args) {  
    ... //check and report argument violations  
    try{  
        _doTheWork(i_args);  
    }catch(final Exception e){  
        multex.Msg.printReport(e);  
    }  
}
```

This will report to `System.err` the exception chain/tree messages using for localization the resource bundle `MsgText` in the current working directory, followed by the stack trace. If you want to use another resource bundle for localization, then use the variant

```
printReport(StringBuffer, Throwable, ResourceBundle)
```

instead, and print out the filled `StringBuffer`, afterwards.

Reporting in Swing Applications

In a Java Swing application, you should report all exceptions by a variant of the static `multex.Swing.report` methods. These report firstly a window with the messages of the exception causal chain/tree. Then pressing on the button **Details** will expand the dialog and show the stack traces of the exception chain/tree, too.

Usually the variant of method `multex.Swing.report` with an owner component should be used, as it will block further input to the owner component or its parent window, until the exception message dialog has been closed by the user.

Centralized exception reporting using `javax.swing.AbstractAction`

Using a simple variant of the *template method* design pattern, you should write your own class, e.g. `UiAction` as a subclass of `javax.swing.AbstractAction`:

```
public class UiAction extends javax.swing.AbstractAction {  
    final void actionPerformed(ActionEvent ev){  
        try{  
            actionPerformedWithThrows(ev);  
        }catch(Exception ex){
```

```
        multex.Swing.report(ev.getSource(), ex);
    }
}
}
```

This class gets the UI component causing the event, uses its parent chain to get the owner frame, and will block it during exception reporting. When using `UiAction` instead of `javax.swing.AbstractAction` you must extend `UiAction` redefining the method `actionPerformedWithThrows` instead of `actionPerformed`.

See an example at the Central Exception Reporting Sample Application, site <http://excrep.berlios.de/>

Last-resort exception reporting

In order to report all uncaught exceptions in a Swing application, which occur during the execution of an UI-event-triggered action, it is sufficient to install one central exception handler using the undocumented system property `sun.awt.exception.handler`, see more in a JGuru discussion [<http://www.jguru.com/faq/view.jsp?EID=427279>].

This undocumented behaviour is wrapped by a MulTEx service. You can implement the interface `multex.AwtExceptionHandler`, and install this class by the method `multex.Awt.setAwtExceptionHandlerClass`. Then any exception propagating up to the AWT/Swing event-dispatch thread will be reported by the handler's method `handle(Throwable)`. In this method you should call a `multex.Swing.report` method.

In order to block input to the GUI application while reporting an exception, it is necessary to locate the application's frame and use it in the central exception handler's call to `report(...)` as the *ownerHook*.

If an application has several frames, and it is not possible to determine automatically, which UI component must be blocked during exception reporting, then you should use an own `UiAction` base class as was described earlier.

Attention: Propagating an exception from the method `actionPerformed` will leave Swing in a dirty state. Sometimes the focus of a menu item will be highlighted in another place, than where Swing assumes it to be. So you should on the one hand install such a last-resort exception reporting, but nevertheless you need an `UiAction` for reporting exceptions in all user-triggered actions.

Reporting in JSP Applications

In a Java Server Pages application, you have several places, where you should report exceptions. These include firstly a JSP error page, which will be called by any unhandled exception. Secondly each JSP page with an input form should contain an error message, if something went wrong during execution of its action. Thirdly, depending on the UI framework used, there is the possibility to report an exception near to the form field, which caused it.

Last-resort exception reporting using an error page

In order to report all uncaught exceptions in a JSP/servlet application, which occur during the execution of an UI-event-triggered action, it is sufficient to install one central error page. See section 2.4.2 *Request Time Processing Errors* of the Sun Java Server Pages Specification about this.

In Tomcat you can do this in the deployment descriptor `web.xml` by an `<error-page>` directive. E.g.:

```
<error-page>
```

```
<exception-type>java.lang.Throwable</exception-type>
<location>/system/errorPage.jsp</location>
</error-page>
```

This means, that any exception, including `Throwable`, will be reported by forwarding to the JSP page `/system/errorPage.jsp`. The error page itself must be marked as such by setting the page directive's `isErrorPage` attribute to `true`, e.g.:

```
<%@page contentType="text/html" isErrorPage="true" %>
```

In such an error page you should report not only the message texts of the exception chain, but also its stack trace and diagnostically useful attributes of the request, session, and application.

See an example at the Central Exception Reporting Sample Application, site <http://excrep.berlios.de/>

Centralized exception reporting with Struts using `org.apache.struts.action.Action`

In Struts 1.3.8 you have the possibility to define a central exception handler using the `<global-exceptions>` element of file `struts-config.xml`. The central exception handler class must implement the interface `org.apache.struts.action.ExceptionHandler`. This works very well and has access to the causing UI component.

A template method for the `execute` methods you need only, if you want to handle other action related issues centrally, e.g. transaction management.

See an example at the Central Exception Reporting Sample Application, site <http://excrep.berlios.de/>

Form Field related Exception Reporting with Struts

One student group even implemented centralized exception reporting for form field related exceptions. In Struts it is possible to `saveErrors` or `saveMessages` indicating the name of the offending field of the form working on. This will place the message in the appropriately marked position in the form.

In the central exception handler for Struts you can specifically handle exceptions, which are related to a form field. E.g. a `FieldvalueExc` could port the identifier of the related field, so that `UiAction` could put the exception message into the good position.

Another idea is to find the place in Struts, where any exception occurring in a `setXxx()` method of a form bean can be caught and reported into the form near to the field `xxx`.

Centralized exception reporting with Java Server Faces

Here is described the solution of the student project TriplePlay in summer 2005.

The JavaServer Faces life cycle (see <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/images/jsfIntro-lifecycle.gif>) is directed by an implementation of the interface `Lifecycle` (in the package `javax.faces.lifecycle`. The reference implementation of Sun is `com.sun.faces.lifecycle.LifecycleImpl`).

When the web application is started, a `LifecycleFactory` (also in package `javax.faces.lifecycle`, reference implementation of Sun is `com.sun.faces.lifecycle.LifecycleFactoryImpl`) is asked to return a `Lifecycle`

object. Normally the standard implementation will be taken. But you have the possibility to change the `LifecycleFactory`, and to return your own `Lifecycle` implementation.

In detail: Here are the two classes, which we have written. You can subclass the reference implementation, or you can completely implement the interface. We take the first approach.

```
package myPackage; //MyLifecycleFactoryImpl.java

import javax.faces.FacesException;
import javax.faces.lifecycle.Lifecycle;

public class MyLifecycleFactoryImpl
extends com.sun.faces.lifecycle.LifecycleFactoryImpl
{

    public Lifecycle getLifecycle(final String i_lifecycleId)
    throws FacesException
    {
        return new MyLifecycleImpl();
    }

}

////////////////////////////////////
package myPackage; //MyLifecycleImpl.java

import javax.faces.FacesException;
import javax.faces.context.FacesContext;

public class MyLifecycleImpl extends com.sun.faces.lifecycle.LifecycleImpl {

    public void execute(final FacesContext io_facesContext) {
        try {
            super.execute(io_facesContext);
        } catch(final FacesException e) {
            // handleExceptions here
        }
    }

}
```

Now we still have to instruct JavaServer Faces, that we want to use our own `LifecycleFactory` implementation instead of the default implementation. This we have to do in the file `faces-config.xml` (a configuration file similar to `struts-config.xml`).

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
    ...
    <factory>
        <lifecycle-factory>myPackage.MyLifecycleFactoryImpl</lifecycle-factory>
```

```
</factory>
...
</faces-config>
```

Having done this, our implementation of the `LifecycleFactory` will be used, which in turn will deliver in the method `getLifecycle` (see before) our own `Lifecycle` implementation.

If there remain questions, I will try to answer them. You can mention my addresses in this little howto, so that future students can ask me, if necessary.

Siamak Haschemi, www.haschemi.org [<http://www.haschemi.org>] E-Mail: [info \(at\) haschemi.org](mailto:info@haschemi.org),

Reporting in background threads

If the `run()` method of a thread terminates due to an uncaught exception, then by default, this thread will die and the stack trace of this exception will be printed to `System.err`.

This default behaviour is not satisfactory. At first, we would like to see the exception message chain, too. And secondly, we would really like to see it in the application's user interface, instead of in a may-be suppressed console output.

Last-resort Exception Reporting

In order to report all uncaught exceptions of all background threads, you must place all the threads in your own `ThreadGroup`. For this `ThreadGroup` you redefine the method:

```
void uncaughtException(Thread t, Throwable e)
```

In this method you can report the exception chain in the format, and to the destination, you want. You should try to report to the user interface (UI), if the thread was created or started by an UI event, e.g. if it implements an action, which would last too long for the user to await its completion.

Since Java 5 there is also such a possibility without the necessity to create thread groups. The method

```
public static void setDefaultUncaughtExceptionHandler(
    Thread.UncaughtExceptionHandler eh
)
```

installs a last-resort exception handler for all thread groups together.

Exception Reporting in long-living threads

But for a long-living thread, usually serving commands or alike, and not triggered by a user action. we cannot tolerate to stop the thread. Thus such a server thread should catch in its command loop in the `run()` method any exception and report it to a logging destination.

Also here it would be useful to inform the user, if any exists, that a required service failed, and where is the appropriate logfile.

How to introduce/use MulTE_x in the API layers

The concepts behind MulTE_x are described in german language in the technical paper `Konzepte.pdf`.

Here are given more practical instructions.

What is a MulTEx exception?

A MulTEx exception is a highly diagnostic exception with a parameterized, internationalizable message text pattern. In contrary the standard `Throwable` of the Java platform does not have an internationalizable message text, and has at most one parameter of type `String`. MulTEx distinguishes between two kinds of exception.

Subclasses of `multex.Exc` serve as business rule exceptions, that is they serve for specifying and implementing the “negative” business rules. In fact, a business rule exception represents the negation of a precondition to a method. When specifying a method, you should specify in its throws clause all exceptions, which the method can guarantee to its caller.

Subclasses of `multex.Failure` serve to indicate to the caller an unexpected failure of a method. Typically they contain a causing exception, which occurred during the method execution. Historically earlier than the JDK 1.4, MulTEx introduced the notion of a causing exception, which in fact results in a chain of exceptions. As this is especially useful in layered software architectures, MulTEx received its name. Nowadays MulTEx even allows a `Collection` of causing exceptions, thus spanning a causal tree.

How to declare a MulTEx exception with an internationalizable message text pattern

The most comfortable way to get an internationalizable message text pattern associated with each exception is the following, usable from MulTEx version 7.1.

You define the message text pattern for each concrete subclass of `Throwable` of which you have the source code. You place the text pattern as the main comment into the Javadoc comment of the exception class before any `@tags`. So the text pattern serves firstly as documentation for this exception, and secondly as text pattern for internationalization.

For example you could define an exception `CreditLimitExc` with a message text pattern, expecting two message parameters, as follows.

```
package org.company.project;

class Account {

    ...

    /**Taking {0} units from your ac
     * count will leave less than your credit limit {1} units on it.
     */
    public static final class CreditLimitExc extends multex.Exc {}

}
```

For `Failures` there is no difference. You only have to declare the exception as a subclass of `multex.Failure`. For example in class `org.company.project.FileUtil`:

```
/**Failure loading file {0} by user {1}*/
public static final class LoadFileFailure extends multex.Failure {}
```

These exception declarations are so simple, as all diagnostic information (causes and parameters) are stored in the MulTEx base class, `Exc`, or `Failure`, respectively. For ease of use they should be initialized using

the generic `create` method in the `MultexUtil` class, rather than the `init`-methods. So the concrete exception class merely serves with its class name as a key to a message text resource bundle.

Collecting the text patterns using the `ExceptionMessagesDoclet`

You can use the doclet `multex.tool.ExceptionMessagesDoclet` to collect all exception message text patterns from the exceptions. You must specify the sources to scan as usually for a doclet. You must pass the output `.properties` file name to the doclet option `-out`.

By executing the doclet on the upper examples the following property definitions will be appended to the given `-out` file:

```
org.company.project.Account$CreditLimitExc = Taking {0} units from your ac\
count will leave less than your creditLimit {1} units on it.
org.company.project.FileUtil$LoadFileFailure = Failure loading file {0} by user {1}
```

As Javadoc removes white space at the end of each comment line, and `java.util.Properties.load` ignores the `\` indicating line continuation and whitespace at the beginning of the next line, it is somehow difficult to make continuation lines with an intuitive spacing. You can either split the long message text inside of a word, as e.g. the word `account` in the example was split, or you can insert a `Properties` escape sequence for whitespace at the end of the line, e.g. a `\t` or a `\u0020`.

Example usage of the `ExceptionMessagesDoclet` from ANT

```
<target name="exception" depends="compile"
  description="Collect all exception message texts in the fb6 software into one
>

  <property name="ExceptionResourcesFile"
    value="${CLASS_DESTINATION}/ExceptionResources.properties"
  />
  <echo file="${ExceptionResourcesFile}" append="false">
    #Generated by ANT-Target _resources. Do not edit.
    multex.MsgText.causeMarker = Ursache:
    #From here are following the message texts for the exceptions in the software
    #They are all collected from the main Javadoc texts of each exception declarat
  </echo>
  <javadoc
    access="private"
    classpathref="compile.classpath"
    doclet="multex.tool.ExceptionMessagesDoclet"
    docletpathref="compile.classpath"
    source="1.6"
    useexternalfile="yes"
    encoding="ISO-8859-1"
  >
    <arg value="-out"/>
    <!--Message texts will be appended to this file.-->
    <arg file="${ExceptionResourcesFile}"/>

    <!--All exceptions in all Java files in this directory will be processed.-->
    <fileset dir="java"/>
  </javadoc>
</target>
```

The former example works as follows. The **echo** element creates a file `ExceptionResources.properties` with a string resource for the key `multex.MsgText.causeMarker`. This defines the german translation for the default cause marker `CAUSE:`. Then the `multex.tool.ExceptionMessagesDoclet` appends all exception message text patterns for all exceptions declared in all Java files in the directory `java` to the `.properties` file. The given encoding is used to interpret the Java source files. For writing to the `-out` file always encoding `ISO-8859-1` is used, as `.properties` files must use it.

How to implement negative business rules by throwing Exc-objects

A method should usually perform its normal specified action. But it often has cases, which are forbidden to be handled like this. In such situations the method should throw an `Exc` object of a specific subclass.

So typically the necessity for throwing an `Exc` can be found out by testing a precondition. You should test one precondition after another, always immediately throwing a specific exception. It is understood, that these exceptions should be specified in the `throws` clause of the method, so that the programmer of the call can consider to change the call or to make a handler for some of them. Here we complete the above example of the bank account.

```
package org.company.project;
import multex.Exc;
import static multex.MultexUtil.create;

class Account {

    final static long creditLimit = -5000;
    long balance = 0;

    /**Transfers the specified amount from this account
     * to the destination account.
     * @throws CreditLimitExc the new balance of the account would become
     * less than the creditLimit.
     */
    public void transfer(final Account destination, final long amount)
        throws CreditLimitExc
    {
        //1. Checking for business rule violations:
        final long newBalance = balance - amount;
        if(newBalance < creditLimit){
            throw create(CreditLimitExc.class, amount, creditLimit);
        }

        //2. Performing modifications:
        balance = newBalance;
        destination.balance += amount;
    }

    /**Taking {0} units from your account\t
     * will leave less than your credit limit {1} units on it.
     */
    public static final class CreditLimitExc extends Exc {}
}
```

```
}
```

Please note here, that for comfortably creating a new parameterized exception object, as we cannot inherit parameterized constructors in Java, we have to call the static generic factory method `MultexUtil.create`, and pass the class of the desired exception, along with its parameters.

How to implement exception chaining by throwing a Failure object

There are some places, where it is recommended to catch an exception, and to wrap it into a parameterized exception with its own message text.

The first is, if the exception is a checked one, and you don't want to specify it in the throws cause of your own method. Then the compiler forces you to catch it. In the catch branch you should usually wrap it.

The other is, even with an unchecked exception, if you want to add abstraction-specific diagnostic parameters, or a message text.

We present here as an example the code for opening a serial port by means of `javax.comm`, as was more detailedly motivated in the german text `Konzepte.pdf`.

```
package x;
import multex.Exc;
import multex.Failure;
import static multex.MultexUtil.create;

class SerialPort {

    ...

    /**Opens the serial port with the given name and the other communication param
    public void open(
        String portName, int baudRate, int databits, int stopbits, int parity
    ) throws NameExc, OpenFailure {
        //1. Checking for business rule violations:
        if(!portName.startsWith("COM")){
            throw create(NameExc.class, portName);
        }

        //2. Performing modifications:
        try {
            this.portName = portName;
            final javax.comm.CommPortIdentifier portId
            = CommPortIdentifier.getPortIdentifier(portName); //NoSuchPortException
            sp = (javax.comm.SerialPort)portId.open(null,0); //PortInUseException
            sp.setSerialPortParams( baudRate, databits, stopbits, parity );
            //UnsupportedCommOperationException
            os = sp.getOutputStream(); //IOException
            is = sp.getInputStream(); //IOException
        } catch (Exception e) {
            ... //free resources
            //wrap exception and parameterize:
            throw create(OpenFailure.class, e,
```

```
        portName, baudRate, databits, stopbits, parity
    );
    } //catch
} //open

/**The serial port name {0} is not allowed.*/
public static final class NameExc extends Exc {}

/**Cannot open the serial port "{0}" with communication parameters portName
 *={1}, baudRate={2}, databits={3}, stopbits={4}, parity={5}
 */
public static final class OpenFailure extends Failure {}

} //SerialPort
```

Here we have the typical case, that the whole modification part of the method body is surrounded by a try-block, in whose catch-branch we catch any `Exception` and propagate it by throwing a `Failure` encapsulating the original exception aside with diagnostic parameters. The caught exceptions include the checked exceptions `NoSuchPortException`, `PortInUseException`, `UnsupportedCommOperationException`, and `IOException`, where commented and any other subclass of `java.lang.RuntimeException`. You always have to pass the causing exception to the first parameter of `MultexUtil.create` after the failure's class.

How to cope with many Failure causes (list exceptions, exception treeing)

There are some cases, when a `Failure` does not have only one cause. This typically occurs, when processing a list of items, each of them being necessary for the success of the complete operation.

For example we want to copy a list of files to a backup directory. Only if we succeed to copy every file, the whole method can be considered successful.

The first approach is

```
/**Copies all files from directory sourceDir to directory destDir.*/
public static void copyFiles1(final File sourceDir, final File destDir){
    try {
        final File[] files = sourceDir.listFiles();
        for(final File file: files){
            copyOneFile(file, destDir);
        }
    } catch (Exception e) {
        throw create(CopyFilesFailure1.class, sourceDir, destDir);
    }
}
```

This method succeeds only, if all files can be successfully copied from the source directory to the destination directory. The first failing copying of a single file will cause the abrupt termination of the complete operation. The resulting `CopyFilesFailure` exception will have as cause only the exception of the first file copy failure.

Sometimes we need the exception of each failed single item. For example a compiler always works like this. It collects error messages, but goes on analyzing the source text. The compilation will fail, if there is at least one error, but all errors will be reported.

If we want our copy files example to handle errors like this, we must collect the individual exceptions before creating the `CopyFilesFailure2`. Then we can pass in this collection to the polymorphic `Object[]` parameters. This is best done by using the constructor `Failure(String, Collection)` with a null `String`. If we want to pass in normal message parameters, too, as in our example `sourceDir` and `destDir`, we should add them to the collection before the exceptions. Then they have a fixed index for the message pattern. The following example with an exception collection is a bit longer, but gives much more diagnostic information.

```
/**Copies all files from directory sourceDir to directory destDir.*/
public static void copyFiles2(final File sourceDir, final File destDir){
    final Collection<Object> parameters = new ArrayList<Object>();
    parameters.add(sourceDir);
    parameters.add(destDir);
    final File[] files = sourceDir.listFiles();
    for(final File file: files){
        try {
            copyOneFile(file, destDir);
        } catch (Exception e) {
            parameters.add(e);
        }
    }
    if (parameters.size()>=2) {
        throw new CopyFilesFailure2(parameters);
    }
}

/**Failure copying files from directory {0} to directory {1}*/
public static final class CopyFilesFailure2 extends Failure {
    public CopyFilesFailure2(Collection i_parameters) {
        super((String)null, i_parameters);
    }
}
```

By the way, giving each exception the potential of a list of causing exceptions, results in spanning a causal tree of exceptions. The standard MulTE_x services for printing an exception will express the causal tree by indenting the lower exceptions by a + sign for each level.

Caution

The standard Java services for printing an exception (`Throwable.toString()` and `Throwable.printStackTrace()`) will ignore causal trees built like this. So be certain, that you have installed centralized exception reporting using MulTE_x services.